


Context-Oriented Algorithmic Design

Bruno Ferreira

Instituto Superior Técnico/INESC-ID, Lisbon, Portugal


bruno.b.ferreira@tecnico.ulisboa.pt

 <https://orcid.org/0000-0001-8227-2648>

António Menezes Leitão

Instituto Superior Técnico/INESC-ID, Lisbon, Portugal

antonio.menezes.leitao@tecnico.ulisboa.pt

 <https://orcid.org/0000-0001-7216-4934>

Abstract

Currently, algorithmic approaches are being introduced in several areas of expertise, namely Architecture. Algorithmic Design (AD) is an approach for architecture that takes advantage of algorithms to produce complex designs, to simplify the exploration of variations, or to mechanize tasks, including those related to analysis and optimization of designs. However, architects might need different models of the same design for different kinds of analysis, which tempts them to extend the same code base for different purposes, typically making the code brittle and hard to understand. In this paper, we propose to extend AD with Context-Oriented Programming (COP), a programming paradigm based on context that dynamically changes the behavior of the code. To this end, we propose a COP library and we explore its combination with an AD tool. Finally, we implement two case studies with our context-oriented approach, and discuss their advantages and disadvantages when compared to the traditional AD approach.

2012 ACM Subject Classification Software and its engineering → Object oriented languages

Keywords and phrases context-oriented programming, algorithmic design, Python

Digital Object Identifier 10.4230/OASIS.SLATE.2018.7

Funding This work was partially supported by national funds through *Fundação para a Ciência e a Tecnologia (FCT)* with reference UID/CEC/50021/2013.

1 Introduction

Nowadays, Computer Science is being introduced in several areas of expertise, leading to new approaches in areas such as Architecture. Algorithmic Design (AD) is one of such approaches, and can be defined as the production of Computer-Aided Design (CAD) and Building Information Modeling (BIM) models through algorithms [21]. This approach can be used to produce complex models of buildings that could not be created with traditional means, and its parametric nature allows an easier exploration of variations.

Due to these advantages, AD started to be introduced in CAD and BIM applications, which lead to the development of tools that support AD programs, such as Grasshopper. However, with the complexity of the models came the necessity of analyzing the produced solutions with analysis tools. For this task, the geometrical models are no longer sufficient, as analysis software usually requires special analytical models, that are different from geometrical models and can not be obtained with import/export mechanisms due to errors. This leads to the production of several models, which have to be kept and developed in parallel, involving different development lines that are hard to manage and to keep synchronized. This complex workflow proves that current solutions are not sufficient [29].



© Bruno Ferreira and António Menezes Leitão;

licensed under Creative Commons License CC-BY

7th Symposium on Languages, Applications and Technologies (SLATE 2018).

Editors: Pedro Rangel Henriques, José Paulo Leal, António Leitão, and Xavier Gómez Guinovart

Article No. 7; pp. 7:1–7:14



OpenAccess Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Some tools like Rosetta [20] address these issues by offering portable AD programs between different CAD applications and, more recently, BIM applications [6] and analysis tools [18]. Nevertheless, Rosetta does not offer a unifying description capable of producing both the geometrical and the analytical models with the same AD program, which can lead to the cluttering of the current program in an effort to reduce the number of files to maintain.

To solve these problems, we propose the use of COP to develop a computational model capable of adapting itself to the required context, which in this case is defined by the requirements of modeling applications and analysis software, allowing the production of different models with a change of context.

1.1 Context-Oriented Programming

COP was first introduced as a new programming approach that takes context into account [7]. According to a more recent depiction of this approach, COP aims to give users ways to deal with context in an explicit way in their programs, making it accessible to manipulate through software with features that are usually lacking in mainstream programming languages [12].

With this approach, users can express different behaviors in terms of the context of the system. This context is composed of the actors of the system, which can determine how the system is used, the environment in which the system is, which can restrict or influence its functionality, and the system itself, whose changes might lead to different responses.

Although there are different implementations of COP, which will be presented later in this paper, according to [12] necessary properties must be addressed by all of them:

- behavioral variation: implementations of behavior for each context;
- layers: a way to group related context-dependent variations;
- activation and deactivation of layers: a way to dynamically enable or disable layers, based on the current context;
- context: information that is accessible to the program and can be used to determine behavioral variations;
- scoping: the scope in which layers are active or inactive and can be controlled.

With these features, layers can be activated or deactivated dynamically in arbitrary places in the code, resulting in behaviors that fit the different contexts the program goes through during its execution. If analyzed in terms of multi-dimensional dispatch [27], it is possible to say that COP has four-dimensional dispatch, since it considers the message, the receiver, the sender, and the context to determine which methods or partial method definitions are included or excluded from message dispatch. These method definitions are used to implement behavioral variations in layers, which can be expressed differently in the different COP implementations. In some, the adopted approach is known as *class-in-layer*, in which layers are defined outside the lexical scope of modules [1], in a manner similar to aspects from Aspect-Oriented Programming (AOP) [17]. In others, a *layer-in-class* approach is used, having the layer declarations within the lexical scope of the modules.

In addition to the base concepts, each implementations has its own operators and strategies that might expand the capabilities of COP. The advantages and disadvantages of each of these implementations will be discussed later in the paper.

1.2 Objectives

The main objectives of this paper are: (1) present and compare the different implementations for COP that have been proposed by the research community, and (2) present two simple case studies that show how COP can be applied in AD. The case studies consist of previously

developed AD programs that we re-implemented with our proposed solution and then used to produce different models according to different context. Finally, the results of our solution are compared to the ones obtained previously.

In the next section, we start by presenting the related work, as well as a comparison between the different implementations of COP.

2 Related Work

In this section, we introduce several paradigms that served as basis for COP, namely AOP and Subject-Oriented Programming (SOP), as well as the more relevant implementations of COP. We end this section with a comparison between these implementations.

2.1 Aspect-Oriented Programming

Most programming paradigms, such as Object-Oriented Programming (OOP), offer some way to modularize the concerns necessary to implement a complex system. However, it is common to encounter some concerns that do not fit the overall decomposition of the system, being scattered across several modules. These concerns are known as *crosscutting concerns*.

AOP was created to deal with these *crosscutting concerns*, introducing ways to specify them in a modularized manner, called *aspects*. *Aspects* can be implemented with proper isolation and composition, making the code easier to maintain and reuse [17]. Using AOP, it is possible to coordinate the *crosscutting concerns* with normal concerns, in well defined points of the program, known as *join points*.

In addition to these concepts, AOP implementations, such as AspectJ, introduce additional ones, such as *pointcuts*, which are collections of *join points*, and *advices*, which are implementations of behavior attached to *pointcuts* [16]. An *aspect* is then a module that implements a *crosscutting concern*, comprised of *pointcuts* and *advices*. AspectJ also offers *cflow* constructs, that allow the expression of control-flow-dependent behavior variations, making it possible to conditionally compose behavior with *if pointcuts* [16].

There are similarities between AOP and COP: both make it possible to define behavior that depends on a condition, which can, in itself, be considered a context. However, while AOP aims to modularize *crosscutting concerns*, this is not mandatory in COP, since the use of *layer-in-class* approaches scatters the code across several modules. In addition, COP allows the activation and deactivation of layers in arbitrary pieces of code, while AOP triggers *pointcuts* at very specific *join points* that occur in the rest of the program [12]. This makes COP more flexible in dealing with behavioral variation.

2.2 Subject-Oriented Programming

SOP is a programming paradigm that introduces the concept of *subject* to facilitate the development of cooperative applications. These are defined by the combination of *subjects*, which describe the state and behaviors of objects that are relevant to them [10].

The *Subjects'* goal is to introduce a perception of an object, such as it is seen by a given application. *Subjects* do so by adding classes, state, and behavior, according to the needs of that application. By doing this, each application can use a shared object through the operations defined for its *subject*, not needing to know the details of the object described by other *subjects* [10].

Subjects can also be combined in groups called *compositions*, which define a composition rule, explaining how classes and methods from different *subjects* can be combined. These can then be used through *subject-activation*, which provides an executable instance of the *subject*, including all the data that can be manipulated by it.

Due to the introduction of all these concepts, SOP offers what is known as *Subjective Dispatch* [27]. This extends the dispatch introduced by OOP, by adding the sender dimension, in addition to the message and the receiver. This was later expanded by COP, which introduces a dimension for context, as mentioned previously.

It is possible to see that, similarly to the other analyzed paradigms, SOP also supports behavior variations in the form of *subjects*. However, if we consider that each *subject* might have different contexts of execution, we need an extra dimension for dispatch, which is what COP offers.

2.3 COP Implementations

COP was proposed as an approach that allows the user to explore behavioral variations based on context. Although this approach introduces concepts such as layers and contexts, each of the implementations address the concepts differently, sometimes due to the support of the host language in which the COP constructs are implemented. In this section we present the different implementations available.

2.3.1 ContextL

ContextL was one of the first programming language extensions to introduce support for COP. It implements the features discussed previously by taking advantage of the Common-Lisp Object System (CLOS) [4].

The first feature to be considered is the implementation of layers, which are essential to implement the remaining features available in ContextL [5]. These layers can be activated dynamically throughout the code, since ContextL uses an approach called Dynamically Scoped Activation (DSA), where layers are explicitly activated and a part of the program is executed under that activation. The layer is active while the contained code is executing, becoming inactive when the control flow returns from the layer activation.

Regarding the activation of multiple layers, it is important to note that the approach introduced in ContextL, as well as in other implementations that support DSA, follows a stack-like discipline. Also, in ContextL this activation only affects the current thread.

By taking advantage of layers, it is then possible to define classes in specific layers, so that the classes can have several degrees of detail in different layers, introducing behavior that will only be executed when specific layers are activated. The class behavior can also be defined with *layered generic functions*. These take advantage of the generic functions from CLOS, and are instances of a generic function class named *layered-function* [5].

In addition, ContextL supports contextual variations in the definition of class slots as well. Slots can be declared as *:layered*, which makes the slot accessible through layered-functions. This introduces slots that are only relevant in specific contexts.

By looking at the constructs implemented in ContextL, it is possible to conclude that behavioral variations can be implemented in specific classes or outside of them. This means that ContextL supports both *layer-in-class* and *class-in-layer* approaches. The former allows the definition of partial methods to access private elements of enclosing classes, something that the latter does not support, since *class-in-layer* specification cannot break encapsulation [1].

Finally, it should be noted that ContextL follows a library implementation strategy: it does not implement a source-to-source compiler, and all the constructs that support the COP features are integrated in CLOS by using the Metaobject Protocol [15].

2.3.2 PyContext and ContextPy

PyContext was the first implementation of COP for the Python programming language. Although it includes most of the COP constructs in a similar manner to other implementations, it also introduces new mechanisms for layer activation, as well as to deal with variables.

Explicit layer activation is an appropriate mechanism for several problems but, sometimes, this activation might violate modularity. Since the behavioral variation may occur due to a state change that can happen at any time during the program execution, the user needs to insert verifications in several parts of the program, increasing the amount of scattered code. To deal with this, PyContext introduces *implicit layer activation*. Each layer has a method *active*, which determines if a layer is active or not. This, in combination with a function *layers.register_implicit*, allows the framework to determine which layers are active during a method call, in order to produce the correct method combination [30].

Regarding variables, PyContext offers contextual variables, which can be used with a *with* statement in order to maintain their value in the dynamic scope of the construct. These variables are called *dynamic variables*. These are globally accessible, and their value is dynamically determined when entering the scope of a *with* construct [30]. In conjunction with specific getters and setters, it is possible to get the value of the variable, change it in a specific context, and then have it restored when exiting the scope of that context. It is important to note that this feature is thread-local.

As for the other features, PyContext does not modify the Python Virtual Machine, being implemented as a library. Layers are implemented using meta-programming, and layer activation mechanisms take advantage of Python's context handlers. As for the partial definition of methods and classes, PyContext follows a *class-in-layer* approach.

More recently, ContextPy was developed as another implementation of COP for the Python language. This implementation follows a more traditional approach to COP, offering Dynamically Scoped Layer activation, using the *with* statement, which follows a stack-like approach for method composition. For partial definitions, ContextPy follows a *layer-in-class* approach, taking advantage of decorators to annotate base methods, as well as the definitions that replace those methods when a specific layer is active [13]. Finally, similarly to PyContext, ContextPy is offered as a library that can be easily included in a Python project.

2.3.3 ContextJ

ContextJ is an implementation of COP for the Java programming language, and one of the first implementations of this approach for statically typed programming languages.

ContextJ is a source-to-source compiler solution that introduces all the concepts of COP in Java by extending the language with the *layer*, *with*, *without*, *proceed*, *before* and *after* terminal symbols [2]. Layers are included in the language as a non-instantiable type, and their definitions follows a *layer-in-class* approach. Each layer is composed of an identifier and a list of partial method definitions, whose signature must correspond to one of the methods of the class that defines the layer. Also, to use the defined layers, users must include a layer import declaration on their program, in order to make the layer type visible.

As for partial method definitions, they override the default method definition and can be combined, depending on the active layers. The *before* and *after* modifiers can also be used in partial method definitions, in order to include behavior that must be executed before and after the method execution. In addition, the *proceed* method can be used to execute the next partial definition that corresponds to the next active layer, allowing the combination of behavioral variations [2].

Regarding layer activation, ContextJ supports dynamically scoped layer activation by using a *with* block. Layers are only active during the scope of the block, and the activation is thread-local. *With* blocks can be nested, and the active layer list is traversed according to a stack approach. This, in combination with the *proceed* function, allows the user to compose complex behavior variations. In addition, it is possible to use the *without* block to deactivate a layer during its scope, in order to obtain a composition without the partial method definitions of that specific layer.

Finally, ContextJ also offers a reflection Application Programming Interface (API) for COP constructs. It includes classes for *Layer*, *Composition*, and *PartialMethod*, along with methods that support runtime inspection and manipulation of these concepts.

2.3.4 Other COP Implementations

The implementations described in the previous sections present the major strategies and features that are currently used with COP. Nevertheless, there are more implementations for other languages, which we briefly describe in this section.

Besides ContextJ, there are other implementations of COP for Java, namely JCop [3] and EventCJ [14], which use join-point events to switch layers, cj [25], a subset of ContextJ that runs on an ad hoc Java virtual machine, and JavaCtx [22], a library that introduces COP semantics by weaving aspects.

There are also COP implementations for languages such as Ruby, Lua, and Smalltalk, namely ContextR [26], ContextLua [31], and ContextS [11] respectively. ContextR introduces reflection mechanisms to query layers, while ContextLua was conceived to introduce COP in games. ContextS follows the more traditional COP implementations, such as ContextL.

In addition, some implementations, such as ContextErlang, introduce COP in different paradigms, like the actor model. ContextErlang also introduces different ways to combine layers, namely per-agent variation activation and composition [24].

Regarding layer combination and activation, there are also implementations that offer strategies that differ from dynamic activation. One example is ContextJS [19] that offers a solution based on open implementation, in which layer composition strategies are encapsulated in objects. These strategies can add new scoping mechanisms, disable layers, or introduce a new layer composition behavior that works better with a domain-specific problem [19].

More recently, Ambience [9], Subjective-C [8], and Lambic [28] were developed. Ambience uses the amOS language and context objects to implement behavioral variations, with the context dispatch made through multi-methods. Subjective-C introduces a Domain Specific Language (DSL) that supports the definition of constraints and the activation of behaviors for each context. Finally, Lambic is a COP implementation for Common Lisp that uses predicate dispatching to produce different behavioral variations.

In the next section we present a comparison between all these implementations, as well as the advantages and disadvantages of using each one.

2.4 Comparison

Table 1 shows a comparison between the analyzed COP implementations.

As it is possible to see, most of the analyzed implementations are libraries, with source-to-source compilers being mostly used in statically typed programming languages. The library implementation has advantages when trying to add COP in an already existing project, since it does not change the language and uses the available constructs. On the other hand, source-to-source compilers, such as ContextJ, introduce new syntax that simplifies the COP mechanics.

■ **Table 1** Comparison between the COP implementations. DSA stands for Dynamically Scoped Activation, LIC for *layer-in-class*, and CIL for *class-in-layer*. Lambic uses predicate dispatching instead of layers, so the last two columns do not apply. Adapted from [23].

	<i>Base Language</i>	<i>Implementation</i>	<i>Layer Activation</i>	<i>Modularization</i>
ContextL	Common Lisp	Library	DSA	LIC, CIL
ContextErlang	Erlang	Library	Per-agent	Erlang Modules
ContextJS	JavaScript	Library	Open Implementation	LIC, CIL
PyContext	Python	Library	DSA, Implicit Layer Activation	CIL
ContextPy	Python	Library	DSA	LIC
ContextJ	Java	Source-to-Source Compiler	DSA	LIC
JCop	Java	Source-to-Source and Aspect Compiler	DSA, declarative layer composition, conditional composition	LIC
EventCJ	Java	Source-to-Source and Aspect Compiler	DSA	LIC
JavaCtx	Java	Library and Aspect Compiler	DSA	LIC
ContextR	Ruby	Library	DSA	LIC
ContextLua	Lua	Library	DSA	CIL
ContextS	Smalltalk	Library	DSA, indefinite activation	CIL
Ambience	AmOS	Library	DSA, global activation	CIL
Lambic	Common Lisp	Library	-	-
Subjective-C	Objective-C	Preprocessor	Global Activation	LIC

As for layer activation, the most common strategy is DSA. However, to increase flexibility, some solutions introduce indefinite activation, global activation, per agent activation or, in the case of ContextJS, an open implementation, allowing users to implement an activation mechanism that best fits the problem they are solving. Although DSA is appropriate for most problems, other strategies might be best suited for multi-threaded applications or problems whose contexts depend on conditions that cannot be captured with the default layer activation approach.

Finally, regarding modularization, it is possible to see that most implementations use the *class-in-layer* or the *layer-in-class* approach. The former allows users to create modules with all the concerns regarding a specific context, while the latter places all the behaviors on the class affected by the contexts. Hence, *class-in-layer* reduces code scattering, while *layer-in-class* simplifies program comprehension. There are implementations that support both approaches, such as ContextL, but usually the supported approach is restricted by the language’s features. Nevertheless, there are cases, such as ContextPy and PyContext, that operate on the same programming language but follow different principles regarding the COP concepts.

All these implementation support the COP paradigm, although they offer different variations of the relevant concepts. Choosing the most appropriate implementation requires a careful examination of their distinct features, and how they help in fulfilling the requirements of the problem at hand.

3 Context-Oriented Algorithmic Design

In this section, we propose to combine COP with AD, introducing what we call Context-Oriented Algorithmic Design. Since it is common for architects to produce several different models for the same project, depending on the intended use (e.g., for analysis or rendering), we define these different purposes as contexts. By doing this, it is possible to explicitly say which type of model is going to be produced.

In addition, we introduce definitions for the design primitives using COP as well. For each primitive, we can define different behavioral variations, depending on the model we

want to produce. For example, since some analysis models require surfaces instead of solids, a primitive function like *Wall* would produce a box in a 3D context and a simple surface in an analysis context.

Finally, since COP allows the combination of layers, we can take advantage of that to combine concepts, such as Level of Detail (LOD) with the remaining ones. This allows more flexibility while exploring variations, since it does not only support the exploration in several contexts, but also the variation of LOD inside the same context, as architects might want lesser detail in certain phases to obtain quicker results.

3.1 Implementation

To test our solution we created a working prototype, taking advantage of Khepri, an existing implementation of AD, and ContextPy to introduce the COP concepts. Khepri is a portable AD tool, similar to Rosetta [20], that allows the generation of models in different modeling back-ends, such as Rhinoceros or Revit, and offers a wide range of modeling primitives for the supported applications. This tool offers a native implementation in Python, a popular programming language among architects, and also a language with a COP implementation, making it easier to extend.

As for the choice of the COP implementation to use, we decided for ContextPy. This implementation is a library, making it easier to include in existing projects and tools, such as Khepri. Also, since AD programs are usually single-threaded, and we can easily indicate the scope to be affected by the context, Dynamic Layer Activation is a good approach to solve the problem. Finally, the code should be as easy to understand as possible, so an implementation with a *layer-in-class* approach would fit our needs, since all the variations are included in the module they modify. All of this is supported by ContextPy.

Having these two tools, we implemented a new library, that extends Khepri, and introduces design elements with contextual awareness. Since the behavioral variations produce the results on the selected modeling tool, this new layer uses Khepri's primitive functions to produce the results. The functions to use depend on the context. For instance, the program uses those that produce surfaces in analysis contexts, and those that produce solids on 3D contexts. For each new modeling element of our library, we define a class with basic behavior and a variation for each possible context, which are defined as layers in the library as well.

Listing 1 shows a simplified definition of the `Slab` object. This definition has a default behavior, and variations for a 3D, 2D, and analysis contexts, which are identified by the layers in the decorators. In each of these methods, Khepri functions are used, in order to produce the results in the modeling tools.

In the next section, we introduce two case studies used to evaluate our approach.

4 Evaluation

For the evaluation of our COP library we started by using an algorithmic model of a shopping mall originally used for evacuation simulations. The model was produced with an algorithmic solution, which we modified to use our library.

We chose this case study because the original implementation required a plan view of the model, which had to be produced in addition to the usual 3D view. To take advantage of the same algorithm, the original developers provided two versions of the shop function which produces each of the shops available in the mall. One is a 2D version that creates lines, and the other is a 3D version that creates solids. In order to switch between them, a couple lines of code were commented and some variables were changed as well. Listing 2

■ **Listing 1** Definition of a simplified Slab object in ContextPy.

```
class Slab:
    def __init__(self, path, thickness):
        self.path = path
        self.thickness = thickness

    @around(a3DLayer)
    def generate(self):
        return extrusion(surface_from(self.path), self.thickness)

    @around(a2DLayer)
    def generate(self):
        return self.path

    @around(anAnalysisLayer)
    def generate(self):
        return surface_from(self.path)
```

■ **Listing 2** Simplified version of the code with the definition of both shops and the selection of one.

```
def shop2d(p, v, l, w):
    ...
    rectangle(...)
    line(...)
    ...

def shop3d(p, v, l, w):
    ...
    cuboid(...)
    ...

#shop = shop2d
shop = shop3d
```

shows a simplified definition of both 2D and 3D versions of the shop function, as well as the commented line of code needed to activate the 2D version, and the active line of code that selects the 3D version.

This approach has several disadvantages, namely the need to change the code when it is necessary to change the type of model, and having to comment and uncomment several lines of code when we want to change from 2D to 3D. Both of these tasks are error prone, since developers might forget to do some of them.

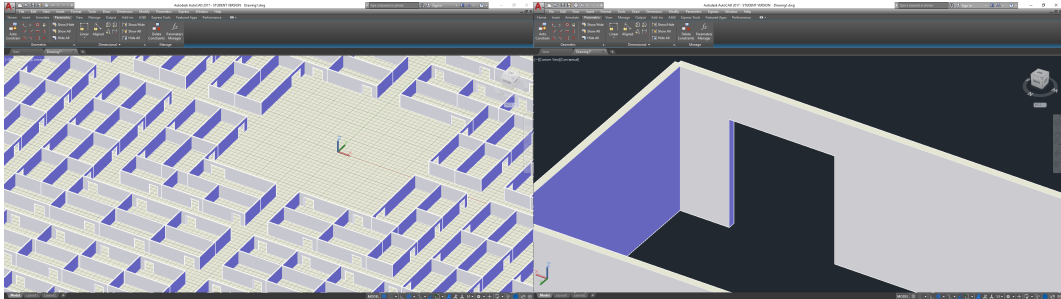
Our COP-based solution eliminates the aforementioned problems. Using our library, we re-implemented parts of the algorithm, namely the shop function. Since we have different implementations for the elements, such as walls, available in different contexts, we do not require two versions of the same function. Listing 3 shows a simplified definition of the COP version of the shop function, using the Wall and Door elements of our library.

■ **Listing 3** Simplified version of the COP version of the `shop` function.

```
def shop(p, v, l, w):
    ...
    w1 = Wall(p1, p2, wall_thickness, wall_height)
    w2 = Wall(p2, p3, wall_thickness, wall_height)
    w3 = Wall(p3, p4, wall_thickness, wall_height)
    w4 = Wall(p4, p1, wall_thickness, wall_height)
    d1 = Door(w4, p5, p6, door_height)
```

■ **Listing 4** Activation of the desired layer.

```
with activelayer(a3DLayer):
    mall(xy(0,0), 100000, 12000, 25000, 7000, 7000, 4)
```



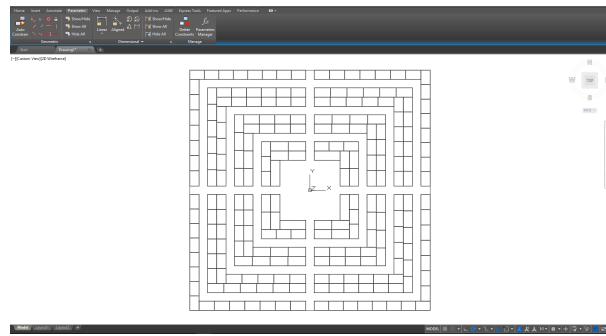
■ **Figure 1** 3D model of the shopping mall, produced with COP in AutoCAD. On the right it is possible to see that the walls are 3D elements.

To shift between contexts, we eliminated the commented lines of code and introduced a `with activelayer` construct, which receives the layer corresponding to the model we want to produce, and the expression that generates the entire shopping mall, as seen in listing 4.

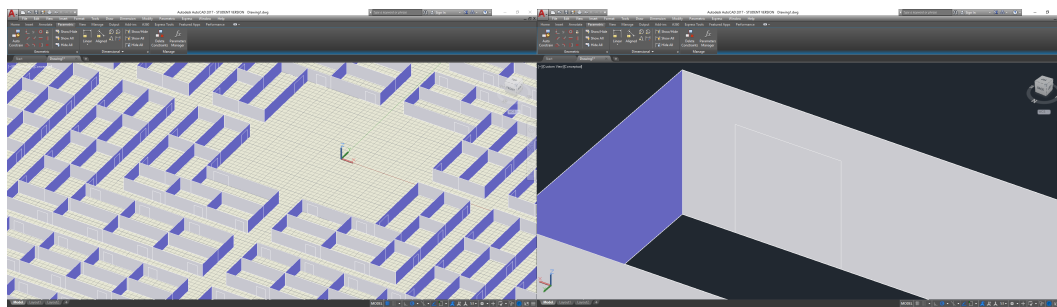
Since we wanted to produce a plan view and a 3D model, and those correspond to layers that we support in the library, we could generate both of them by introducing `a3DLayer` or `a2DLayer` as arguments of the `with` construct. The results can be seen in figures 1 and 2. In addition, since we support a layer that produces only surfaces for analysis purposes, namely radiation analysis, we were able to produce another model simply by changing the context. By using `anAnalysisLayer` as argument for the `with` construct, we produced a model for analysis (visible in Figure 3) without any changes to the algorithm.

With our solution, we were able to reduce the code that produces the models and introduce a more flexible way to both change the context and produce different views of the model. This did not required any additional functions, except the `with` construct, as seen in listing 4. In addition, by simply introducing new contexts, our algorithms are capable of producing new models without any changes.

We also re-implemented another case study, which generated a private house. Due to energy requirements, the generated house had to be analyzed regarding solar radiation. Since this analysis only required part of the model and the substitution of all solid elements with surfaces, the original code had several functions that produced each of the required variations, similarly to the previous example.



■ **Figure 2** 2D model of the shopping mall, produced with COP in AutoCAD.



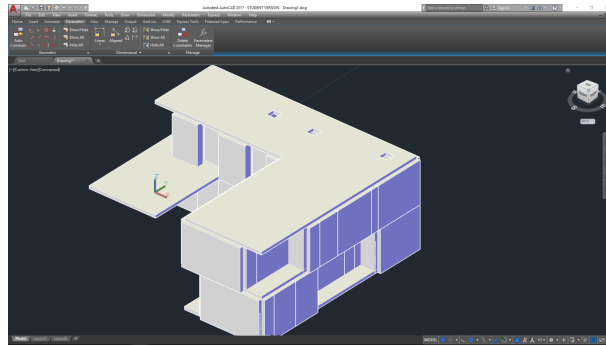
■ **Figure 3** Model of the mall produced for analysis, in AutoCAD. All the walls were replaced with surfaces, as it is possible to see in more detail on the right.

To simplify the algorithmic description of the house, we introduced two new COP layers in the program: (1) the `floorAnalysisLayer`, which only generates the ground floor and replaces elements with surfaces; and (2) the `fullModelLayer`, which produces the complete model in 3D. Both of these layers are used in combination with the layers described in the previous example, in order to define a contextual-dependant definition of the house function. When the house function is used with `fullModelLayer`, the `a3DLayer` is activated and all the elements of the house are produced in 3D, as seen in figure 4. If the `floorAnalysisLayer` is used instead, `anAnalysisLayer` is activated and all the elements are produced as surfaces. In addition, the layer only generates the necessary elements for analysis, as seen in figure 5.

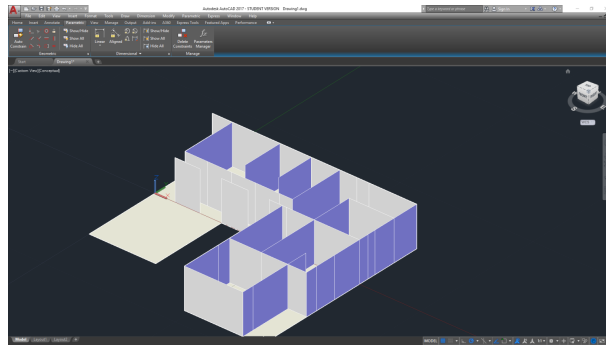
This case study illustrates another advantage of our proposed approach, which is the ability to change layers dynamically during the program execution. This feature offers more flexibility to developers, allowing the production of simplified models for analysis purposes, and more complex ones for 3D visualization. With COP this can be achieved by simply defining and changing layers, instead of using multiple functions with complex conditional statements.

5 Conclusions and Future Work

Currently, algorithmic approaches are used in Architecture to create complex models of buildings that would otherwise be difficult to produce. Moreover, AD also simplifies and automates tasks that were error-prone and time-consuming, and allows an easier exploration of variations. Nevertheless, when architects need to produce different views of the same model, the algorithmic representations must be adapted, creating different versions that increase maintenance efforts.



■ **Figure 4** 3D model of a house, produced with COP in AutoCAD.



■ **Figure 5** Simplified 3D model of the house for analysis, produced with COP in AutoCAD.

In this paper, we explore the combination of AD with COP, a paradigm that dynamically changes the code's behavior depending on the active context. There are multiple implementations of COP for different programming languages, namely ContextL, ContextJ, and ContextPy, among others, all of which have different features and advantages.

In our solution, we took advantage of ContextPy, a library implementation for the Python programming language, that uses DSA, and a *layer-in-class* approach. All these features fit the needs of AD problems, and the use of Python simplifies the introduction of COP in existing tools, such as Khepri.

To test our solution, we used our COP library in existing AD programs that produced models for different types of analysis. The programs included multiple definitions of the same functions to produce different views of the model, which were activated by commenting and uncommenting code. Both programs were re-implemented with COP, which eliminated the multiple definitions and the commented code. This allowed the production of the models for several different contexts without additional changes in the program.

With these examples, we can conclude that COP can be combined with AD and it can be useful when exploring different views of the models, which require different behaviors from the same program.

As future work, we will continue to expand our library with more building elements. We will also introduce more contexts, giving users more layers for the production of different kinds of models.

In addition, we will explore the combination of layers. One idea is to explore a LOD layer in combination with the others, in order to produce simpler models in an exploration phase, and more complex ones in later stages of development.

Finally, we want to research the methodology that users should follow to use our approach, as well as the features that should be included in a development environment to simplify the use of this approach by non-expert users, such as architects.

References

- 1 Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, pages 6:1–6:6, 2009. doi:10.1145/1562112.1562118.
- 2 Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Information and Media Technologies*, 6(2):399–419, 2011.
- 3 Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Software Composition*, volume 6144, pages 50–65, 2010.
- 4 Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *ACM-SIGPLAN Notices*, 23, 1988.
- 5 Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextL. In *Symposium on Dynamic languages*, pages 1–10, 2005.
- 6 Sofia Feist, Guilherme Barreto, Bruno Ferreira, and António Leitão. Portable generative design for building information modelling. In *Living Systems and Micro-Utopias: Towards Continuous Designing - 21st International Conference of the Association for Computer-Aided Architectural Design Research in Asia*, pages 147–156, 2016.
- 7 Michael L. Gassanenko. Context-oriented programming. In *EuroForth'98 Conference*, 1998.
- 8 Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In *International Conference on Software Language Engineering*, pages 246–265, 2010.
- 9 Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object system. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008. doi:10.3217/jucs-014-20-3307.
- 10 William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Notices*, 28(10):411–428, 1993. doi:10.1145/167962.165932.
- 11 Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407. Springer, 2008. doi:10.1007/978-3-540-88643-3_9.
- 12 Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.
- 13 Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic contract layers. In *ACM Symposium on Applied Computing*, pages 2169–2175, 2010.
- 14 Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Designing event-based context transition in context-oriented programming. In *2nd International Workshop on Context-Oriented Programming*, pages 2:1–2:6, 2010. doi:10.1145/1930021.1930023.
- 15 Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.
- 16 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327–354, 2001.

- 17 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242, 1997.
- 18 Ant[on]io Leit[ã]o, Renata Castelo Branco, and Carmo Cardoso. Algorithmic-based analysis - design and analysis in a multi back-end generative tool. In *Protocols, Flows, and Glitches: 22nd CAADRIA Conference*, pages 137–147, 2017.
- 19 Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, 2011.
- 20 Jos[é] Lopes and Ant[on]io Leit[ã]o. Portable generative design for CAD applications. In *31st Conference of the Association for Computed Aided Design in Architecture*, pages 196–203, 2011.
- 21 Jon McCormack, Alan Dorin, and Troy Innocent. Generative design: a paradigm for design research. *Proceedings of Futureground, Design Research Society*, 2004.
- 22 Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. JavaCtx: seamless toolchain integration for context-oriented programming. In *3rd International Workshop on Context-Oriented Programming*, pages 4:1–4:6, 2011. doi:10.1145/2068736.2068740.
- 23 Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012. doi:10.1016/j.jss.2012.03.024.
- 24 Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: introducing context-oriented programming in the actor model. In *11th International Conference on Aspect-oriented Software Development*, pages 191–202, 2012. doi:10.1145/2162049.2162072.
- 25 Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *ACM Symposium on Applied Computing*, pages 1944–1951, 2009.
- 26 Gregor Schmidt. ContextR & ContextWiki. Master’s thesis, Hasso-Plattner-Institut, Potsdam, 2008.
- 27 Randall B Smith and David Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996.
- 28 Jorge Vallejos, Sebasti[á]n Gonz[á]lez, Pascal Costanza, Wolfgang De Meuter, Theo D’Hondt, and Kim Mens. Predicated generic functions. In *Software Composition*, volume 6144, pages 66–81, 2010.
- 29 Ramon van der Heijden, Evan Levelle, and Martin Riese. Parametric building information generation for design and construction. In *Computational Ecologies: Design in the Anthropocene - 35th Annual Conference of the Association for Computer Aided Design in Architecture*, pages 417–429, 2015.
- 30 Martin Von L[ö]wis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM, 2007.
- 31 Benjamin Hosain Wasty, Amir Semmo, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. ContextLua: dynamic behavioral variations in computer games. In *2nd International Workshop on Context-Oriented Programming*, pages 5:1–5:6. ACM, 2010. doi:10.1145/1930021.1930026.